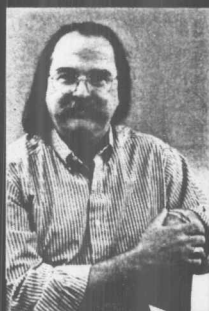


Speed Racer

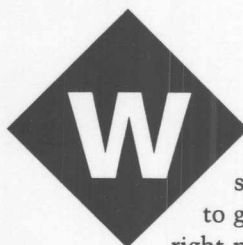
FROM THE BENCH

Jeff Bachiochi

Virtual Speed with the SX



In the hustle of modern life, speed is essential. Jeff knows this. He thinks there is no time like the present to cover the inner workings of today's fastest microcontrollers.



We are driven by speed. We expect to get what we want right now. Give me food now—instant breakfast, fast food lunch, and pizza delivery for supper. Give me weather now—plan the whole week based on a 10-s forecast. Give me phone service now—call from anywhere, anytime, but don't put me on hold. Give me Internet now—bring up this site now (what good is a 500-MHz processor if it still takes thirty seconds to download site data). Get me there now—raise the speed limit, use the drive-through, and pay at the pump. Road rage is spreading because we hate to wait. If we can't keep pace, it all starts to crumble.

Maybe we should just let it crumble a bit. Just enough to let you feel like you're in charge. When you do, the food will taste better, the weather will seem more predictable, and you might even figure out what day of the week it is before Friday comes!

If you work with processors, you know that execution speeds continue to increase. Some of the latest technologies are now using internal PLLs to create execution clocks faster than the crystals that run them. One advantage to this internal speed, beyond the obvious, is less EMI,

which is a serious threat to any product's acceptance. A disadvantage of PLL clocks is jitter (edge accuracy). Whatever the potential disadvantage, programmers usually agree, more speed is better.

TALKING 'BOUT MY GENERATION

Do today's speeds lend themselves to the generation of waveforms? Certainly, the creation of a TTL output square wave isn't a big deal. Merely setting and clearing an output bit over and over again doesn't take a lot of overhead. Let's use 1 μ s as a typical instruction cycle. This setting and clearing would result in a 500-kHz square wave. There will be a glitch in the timing as you try to jump back to the loop unless you filled the micro's code memory with set, clear, set, clear allowing the eventual wrap to automatically bring you back to the beginning.

All of this assumes that the processor did not require any kind of initialization code. So as it stands, the shortest loop would be a bit slower than just set, clear, set, clear. It is necessary to introduce a slight delay between the set and the clear. The delay should have the same number of cycles as the jump instruction needs after the clear.

- Start—set bit, 1 instruction cycle
- Nops—2 instruction cycles
- Clear bit—1 instruction cycle
- Jump start—2 instruction cycles

The number of nops depends on the number of instructions cycles in the jump command to keep the square wave symmetrical (50% duty cycle). This number can vary from two to many, especially if the instructions are pipelined. The best case is six instruction cycles per loop (166-kHz square wave).

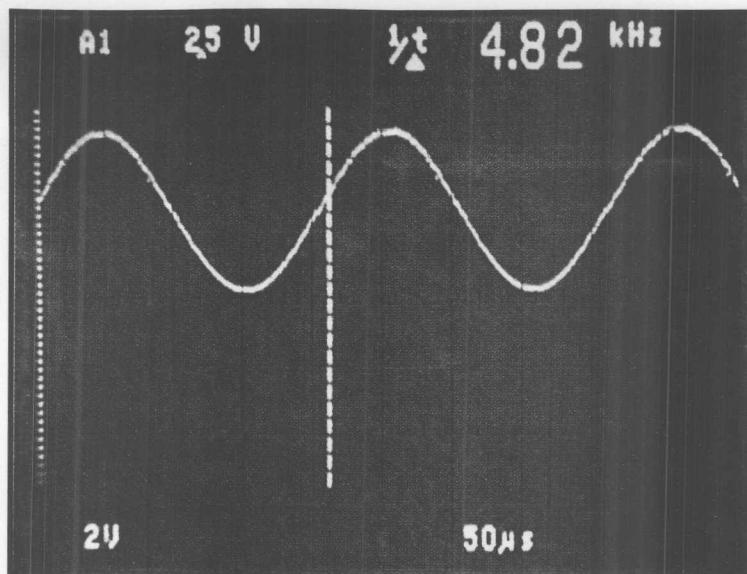
Now, let's suppose that a single frequency output isn't good enough. You want to have the ability to change the frequency. You might choose to use a number of input bits to select the frequency. Reading a byte value from an input port certainly is quick, and it gives you 256 different frequencies.

To place variable delays between the set and clear commands you might choose to place 256 nops between them and use a jump+offset command to jump to specific points within the nop delays. The byte read from the input port could provide the offset of the jump. So, you add the port read command, which adds another instruction cycle to the loop (actually it adds double because you need to balance the added instructions on both half cycles of the waveform). The loop is now a minimum of eight instructions long, for a maximum frequency of 125 kHz.

TIME IS ON MY SIDE

Using the processor's timer function is a much better idea. This method will also allow for longer delays if it has a 16-bit timer available. However, an 8-bit timer can also be used. Timers are normally incremented by the instruction clock. The timer generates an interrupt when the time increments (or decrements) past its maximum or minimum count and overflow (or underflow) of the counter. If the timer is allowed to count without the user altering the count value, it will interrupt again in 256 counts (or 65536 counts for 16-bit counters).

Photo 1—This is the output produced by the SX processor. On close inspection, you can see the actual output steps produced by the R2R ladder. This DAC was constructed with 5% resistors.



The time between interrupts can be adjusted by altering the counter value anytime after an interrupt yet before the next interrupt. In fact, careful attention must be paid to updating the count, especially with 16-bit counters, as the count may be incorrectly incremented if the low byte of the count overflows into the high byte before the high byte is updated.

Of course, the timer can be used without interrupts. Some processors don't have interrupts. Without interrupts you are required to stay in a loop waiting for the overflow to occur.

Although using the timer can be easier than jump+offset, without interrupts it's a pain.

The advantage of using interrupts is that the set and reset commands can be part of the interrupt background routine allowing other things to take place in the foreground, but interruptable (at a lesser priority). An important pitfall to avoid when using the timer is that any count placed into the timer must not overflow before the interrupt routine exits. Overflows, which take place during a timer interrupt, will cause another interrupt immediately upon exiting. In this case not only is the timing wrong (late), but no other code will ever be executed except for the interrupt routine.

There are several ways of preventing this from happening. If a prescaler is available, you can choose a prescale divisor, which will only send the timer an increment every other instruction cycle (or other multiple thereof). This enables a few more instructions to execute before incrementing the timer and essentially slows down the timer, creating a longer tick. The longer the tick, the greater the difference in minimum frequency change.

Alternately, the timer can be turned off while the count is loaded and reenabled just before the ret i. This method adds a few instruction cycles to the delay, but it keeps the timing accurate. Still, the minimum timing (maximum frequency) is the

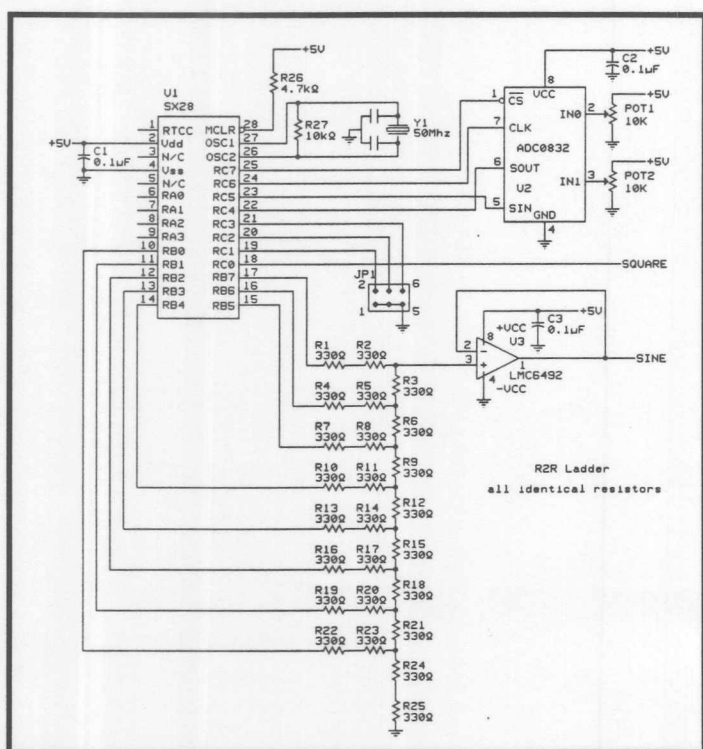


Figure 1—The R2R ladder is a programmable current source into R_{out} . V_{out} can vary from 0 volts to $V_{cc}/2N$ where N is the number of output bits.

maximum code path through the interrupt. In this case, save registers turn off the timer, complement the output bit, reload the timer, turn on the timer, restore registers, and exit. This might be as little as 17 instructions/half cycle (~30 kHz).

You might be getting the picture about now. It doesn't take long before the steps necessary to perform a function begin to detract from the original purpose (to program a variable high-frequency square wave oscillator).

Imagine now that we also want other things like PWM on the square wave output or alternately, a sine or triangle output. If we choose to create a sine wave using 256 discrete steps, that would (at the absolute minimum) limit the output frequency to 111 Hz. If we could find a micro with a faster instruction cycle certainly that would help things, right?

BIT-BALL WIZARD

I received my first samples of Scenix Semiconductor's first 50-MHz microcontroller in early 1998. Their first flash memory-based micros were poised to pounce on Microchip's low-end micros. Flash memory makes for quick development. Their idea at Scenix is to keep the price lower by eliminating all the on-chip hardware peripherals.

Scenix figures if they give you enough speed, you can create virtual peripherals on an as-needed basis.

Listing 1—In the timer overflow interrupt routine, the value of A/D channel #1 determines how many interrupts must occur before the square wave output bit is flipped.

```

org      $000
TMROVF   mov    RTCC, #$E0      ; (2)
TMRO_SQR test   POT1           ; (1)
          jz     TMRO_SQRO      ; (2,4 skip)
          dec    POT1           ; (1)
          jmp    TMRO_SQRX      ; (3)
TMRO_SQRO mov    POT1, CNT1      ; (2)
          inc    CNT            ; (1)
          movb   CNT.0, RC.0     ; (4)
TMRO_SQRX reti                  ; (3)

```

These micros do include an 8-bit timer with an interrupt. Additionally, interrupts have a hardware context save/restore to automatically take care of some necessary housecleaning. There is no timer overflow flag to poll, so if you don't use the interrupt service routine, you must continuously read the timer to determine if it has rolled over.

Because the SX's timer can't be stopped and started, turning off the timer isn't an option. Therefore, in order to prevent reload values from timing out before the routine has exited, the reload value must be greater than the longest path through the routine.

In Listing 1, we are trying to get our square wave output to be as fast as possible, yet still be able to vary its frequency.

The longest path through this interrupt routine requires 20 in-

struction cycles. The timer must be loaded with a value, which will not cause a rollover until after the interrupt exits. If you want any real work done before the next interrupt comes along, you must adjust this value appropriately. Doubling the 20-instruction count to 40 would balance the execution to 50% interrupt, 50% other work. Adjusting the reload value sets the tick time (minimum duty cycle) of the output waveform. This is the highest possible frequency, so it makes sense to keep this tick as fast as possible. Here I use a value of 32 (eight cycles have passed by the time I get to reload the count).

Producing the output waveform is handled by the interrupt routine, so what else is left? If we wish the output frequency to be variable, we need to collect a control value from somewhere. Although an 8-input port allows for 256 possibilities, I wanted to use a potentiometer to make tuning easier than plugging and unplugging jumpers. I decided to add a simple serial 8-bit A/D (see Figure 1), because producing a virtual A/D really requires use of the RTCC (or at least no interrupting) and an external capacitor to charge and discharge.

A small 8-pin dual-channel A/D is available from a number of manufacturers (e.g., National's ADC0832). The nice thing about using these serial devices is that the clocking is asynchronous. The main loop, which reads the A/D, can be interrupted by the RTCC interrupt without causing problems in getting the A/D's converted value (see Listing 2). The 8-bit value read is used as the reload value of the tick counter (POT1).

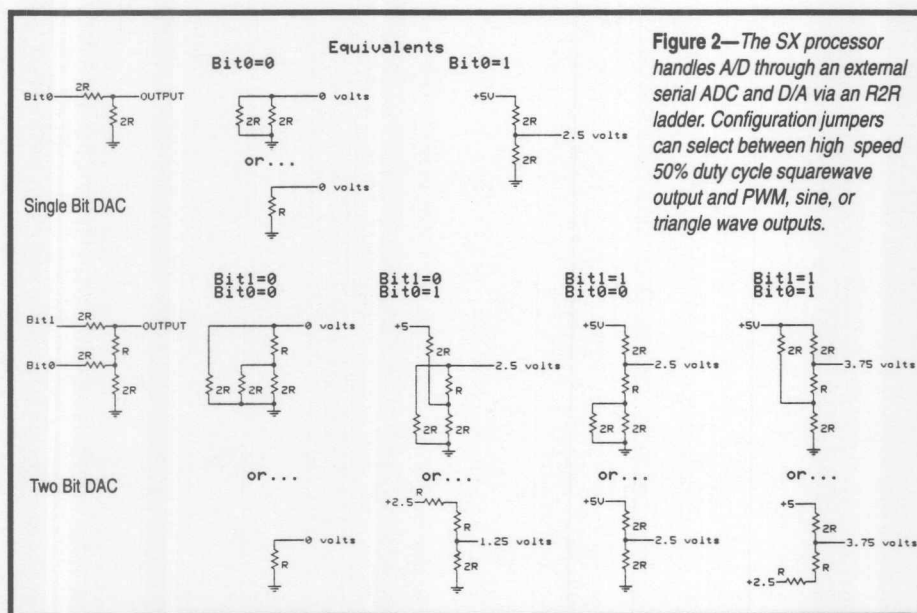


Figure 2—The SX processor handles A/D through an external serial ADC and D/A via an R2R ladder. Configuration jumpers can select between high speed 50% duty cycle squarewave output and PWM, sine, or triangle wave outputs.

Listing 2—The second A/D channel adds a "fine" adjustment to extend the half cycle count to a 16-bit value.

```

org      $000
TMROVF   mov     RTCC,#$E0      ; (2)
TMRO_SQR test    POT1           ; (1)
          jz      TMRO_SQR      ; (2,4 skip)
          dec     POT1           ; (1)
          jmp     TMRO_SQRX      ; (3)
TMRO_SQR test    POT2           ; (1)
          jz      TMRO_SQR0      ; (2,4 skip)
          dec     POT1           ; (1)
          dec     POT2           ; (1)
          jmp     TMRO_SQRX      ; (3)
TMRO_SQR0 mov     POT1,CNT1      ; (2)
          mov     POT1,CNT2      ; (2)
          inc     CNT            ; (1)
          movb    RC.0,CNT.0     ; (4)
TMRO_SQRX reti                     ; (3)

```

I can add a second pot because I have two A/D inputs. This little change adds a second A/D channel to increase the tick counter value from an 8-bit to a 16-bit value. It adds eight execution cycles to the interrupt loop, but because I didn't readjust the reload value, all the timing remains the same. This 16-bit count creates a 50% square wave variable from 625 kHz down to 10 Hz. Ultimately this maximum speed is reached, thanks to the 50-MHz clock and the 20-ns execution cycle of the SX micro.

The next step brings us to PWM square waves. We've already got a couple of 8-bit inputs to the processor, so let's set up one as frequency and

one as duty cycle. Right off the bat, the maximum frequency is going to go down by a factor of 256 (the control range of the duty cycle parameter). So the most we could hope for is about 2500 Hz. The math necessary to produce a value for the duty cycle on or off time based on the total cycle time is eliminated with this scheme.

The 8-bit frequency value (POT1) is used as a loop-count value. Every time the interrupt loop is entered this value is decremented until it reaches zero. This produces the variable-frequency portion of the output. However, once it reaches zero, an 8-bit counter (CNT) is incremented. Whenever this counter reaches zero, the square wave

output bit is cleared. Although this reduces the frequency by 256, it does divide the selected frequency set by POT1 into 256 equal pieces.

Because the duty cycle value (POT2) is also 0-255, this value can be used to directly control duty cycle without any math. The off time is this value in relation to the counter (CNT). Remember that the output bit is cleared when CNT = 0? In the same way, the output bit is set when the duty cycle decrements to zero. So, the output bit goes low once every 256 counts (CNT) and goes high at one of those counts, when POT2 decrements to zero. Again, most of the real work is done in the interrupt routine (see Listing 3).

The maximum number of execution cycles through the interrupt loop is now 29. We still don't have to increase the minimum interrupt time (RTCC reload value). Because our operation is performed on each half cycle of 625-kHz/256 frequency, the maximum output frequency is ~5 kHz and not 2.5 kHz as suggested previously. Duty cycle is frequency independent and adjustable in <0.5% increments.

You might wish to trade off duty cycle increment size for maximum frequency. Reducing the duty cycle resolution to 128 allows the maximum frequency to go up by a factor of two resulting in ~10 kHz. I chose to use 256 because it greatly simplifies the programming, and it leads into the next section quite nicely.

SURFIN' USA

Certainly there must be life beyond square waves. In fact, most signal generators provide sine and triangle outputs in addition to square waves. One of the easiest ways of producing sinewaves is to provide a PWM output from which you could filter out most of the harmonics, leaving a clean (as clean as they get) sinewave. I don't want to use this method, because it doesn't lend itself well to a varying frequency output.

The more direct approach is to use a DAC. The cycles necessary for writing (rapidly enough) to a serial DAC would greatly reduce the

Listing 3—In this PWM implementation, the A/D channel #2 value becomes a count of how long the output will remain low for each cycle.

```

org      $000
TMROVF   mov     RTCC,#$E0      ; (2)
          jnb     PC.CFG,TMRO_SQR ; (2,4 jump)
TMRO_PWM test    POT1           ; (1)
          jz      TMRO_PWM0      ; (2,4 jump)
          dec     POT1           ; (1)
          jmp     TMRO_PWMX      ; (3)
TMRO_PWM0 inc     CNT            ; (1)
          test    POT2           ; (1)
          jz      TMRO_PWM00      ; (2,4 jump)
          dec     POT2           ; (1)
          jmp     TMRO_PWM000     ; (3)
TMRO_PWM00 mov    POT2,CNT2      ; (2)
          setb    RC.0           ; (1)
TMRO_PWM000 test   CNT           ; (1)
          jnz     TMRO_PWM0000    ; (2,4 jump)
          clrb    RC.0           ; (1)
          mov     POT2,CNT2      ; (2)
          mov     POT1,CNT1      ; (2)
TMRO_PWMX reti                     ; (3)

```

maximum frequency available here even more than the 5 kHz we presently have. So, I don't want to use this method either.

A third possibility uses an R2R ladder as a parallel DAC by writing 8-bit values directly to an output port. Using either DAC method requires some computations or a lookup table for determining the appropriate output values. As if you haven't figured it out already, notice that I've been paving the way for a 256-byte lookup table by the way I've designed the PWM square wave interrupt routine.

The CNT variable which was used to divide the square wave into its 256 possible duty cycle points can now become the offset into a lookup table. Instead of getting the off-time from POT2 for the PWM square wave output, in this method POT2 is not needed, since the table will supply all of the necessary wave shape data. The value returned from the table is simply placed into Port B's output register. Where did this data come from?

Attempting to calculate the necessary sine data on the fly at each new degree point within the 360° of a cycle would again be too time consuming. I wrote a few QBASIC program lines on my PC to calculate the hex values for each table entry. Although I could have had this program create a file properly formatted for direct copying into my source code, I opted for just a simple program to print a paper list of the table offset and entry data.

I entered this data by hand into the source code. However, I did need to make a small adjustment. Using Parallax's SK-KEY development system I noticed that the `jmp PC+W` command is actually `jmp PC+W+1` and it can jump to the first 256 bytes of each 512-byte page. Because the actual jump command must reside in the same 256 bytes, when `W = $FF`, the jump will actually end up back on top of itself (because it can't roll into the upper 256 bytes). To prevent this from playing havoc with program execution, I test for `W=$FF` and force it to `W=$00`. This means that the offset

data at \$00 will in fact be used twice for each cycle.

The data presented to Port B is the same data that would be passed to an 8-bit hardware DAC. I create a DAC by using an R2R ladder on these output pins and following the ladder with an OP-AMP to buffer the output signal. For more detail, see the analysis of how the R2R ladder works in Figure 2. Each bit of the R2R ladder creates a voltage divider capable of controlling the voltage difference between its output pin and the next lower bit's output.

The higher the number of bits in the resistor DAC, the closer to V_{CC} the output can reach. With a single bit the R2R output switches between $\frac{1}{2} V_{CC}$ and ground. With two bits, the R2R output can reach $\frac{3}{4} V_{CC}$ with all output bits high. At 8-bits the R2R output will reach 99.6% of V_{CC} with an LSB of 19.5 mV (sound familiar?) The actual output impedance will always equal R. Take a look at the actual output waveform produced by the SX chip in Photo 1.

From this you can see that adding other special waveforms is no more difficult than pointing to a different table. The number of possible waveforms is limited only by the amount of space available for the tables. The total space needed for the code is less than 256 bytes (not counting any tables). You might find that you could develop algorithms, which require less table space.

You can do sine waves of the same resolution with a table of only 64 bytes. However, figuring out which quadrant you're in and in which direction you would need to read through the table, would seriously reduce the maximum frequency. Often simpler is better, since you'd be trading fanciness for crucial time. Note that many compilers can be optimized for either maximum speed or minimum code size.

SLOW RIDE

When I look back at what I started with (a 50-MHz processor with a 20-ns execution time), I can't help but feel that there is something basically wrong here. Although I could get

some fairly fast square wave outputs (>600 kHz), when it comes to generating some of the other useful basic waveforms, speed deteriorates quickly. There is a 10,000:1 difference between the execution cycle time and the maximum sine wave frequency output (based on my resolution criteria).

It's true I haven't yet delved too deeply into cutting every corner and tightening up the code to its absolute minimum execution times. I thought it would be better to work through this based on clarity as opposed to absolute maximum attainable speed. Besides, no matter what I come up with, I'm sure there are many of you out there who will program cycles around me. You don't see many waveform generators based on a microcontroller, and I guess this is for several good reasons. Achieving complex variable high-frequency waveforms takes precious cycle time.

It's not very often that the ideas I investigate actually end up as a piece of useful equipment. But I seem to always have the need for a simple waveform generator. After all, 10,000:1 sounds pretty good when compared to the odds of winning the lottery. ☒

Jeff Bachiochi (pronounced "BAH-key-AH-key") is an electrical engineer on Circuit Cellar's engineering staff. His background includes product design and manufacturing. He may be reached at jeff.bachiochi@circuitcellar.com.

SOURCES

SX-28

Scenix Semiconductor, Inc.
(408) 327-8888
Fax: (408) 327-8880
www.scenix.com

SX-KEY

Parallax, Inc.
(916) 624-8003/8333
Fax: (916) 624-8003
www.parallaxinc.com

ADC0832, LMC6492

National Semiconductor
(408) 721-5000
Fax: (408) 739-9803
www.national.com